УДК 381

A TURING MACHINE IS JUST A FINITE AUTOMATON WITH TWO STACKS: A COMMENT ON TEACHING THEORY OF COMPUTATION

Kreinovich Vladik, PhD, Professor,
vladik@utep.edu
Kosheleva Olga, PhD, Associate Professor,
University of Texas at El Paso
olgak@utep.edu

Abstract. Traditionally, when we teach Theory of Computations we start with finite automata, we show that they are not sufficient, then we switch to pushdown automata (i.e., automata-with-stacks). Automata-with-stacks are also not sufficient, so we introduce Turing machines. The problem is that while the transition from finite automata to automata-with-stacks is reasonably natural, Turing machine are drastically different, and as a result, transition to Turing machines is difficult for some students. In this paper, we propose to solve this pedagogical problem by emphasizing that a Turing machine is, in effect, nothing else but a finite automaton with two stacks. This representation make transition to Turing machines much more natural and thus, easier to understand and to learn.

Keywords: teaching Theory of Computation, finite automata, pushdown automata, Turing machines

How we currently teach theory of computation. One of the important topics in theory of computation is understanding different types of computational models; see, e.g., [1].

Usually, the class starts with finite automata, the simplest possible computational device. After studying properties of finite automata, we then provide an example of a language that cannot be recognized by a finite automaton. A usual example of such a language is the language

$$L = \{a^n b^n, n = 0, 1, \ldots\},$$

i.e., the language consisting of an empty string, a string ab, a string aabb, aaabbb, etc.

To recognize this language, it turns out to be sufficient to equip the finite automaton with a stack, i.e., a first-in-first-out device. In computer science terms, a stack is natural – this is how memory is allocated in a computer. Thus equipped finite automata are known as pushdown automata. After studying properties of pushdown automata, we then provide an example of a language that cannot be recognized by a pushdown automaton. A usual example of such a language is the language

$$L = \{a^n b^n c^n, n = 0, 1, \ldots\},$$

i.e., the language consisting of an empty string, a string abc, a string aabbcc, aaabbbccc, etc.

To recognize this language, we need to introduce a new concept: a Turing machine. Turing machines are already sufficient to perform all possible computations.

Turing machines are, to some extent, a pedagogical problem. The transition from finite automata to finite automata with stacks (i.e., to pushdown automata) is reasonably natural. The notion of a stack is well known and actively used in computer science, and even students who have not yet studied stacks know that they exist – since a typical error message in recursive programs is "stack overflow". It is very clear that finite automata are a particular case of pushdown automata: indeed, to get a finite automaton, it is sufficient to simply ignore the stack: do not push anything into it, do not pop anything from it, and you will get the usual finite automaton.

In contrast, the transition from pushdown automata to Turing machine is not intuitive at all. Instead of building upon the previous concept of a pushdown automaton (as we did in the previous transition), we simply introduce a completely new concept, a concept so different from the previous one that even proving that every pushdown automaton can be represented by a Turing machine is not easy.

Because of this, the transition to Turing machines is a stumbling block for many students.

Our solution to this pedagogical problem: main idea. In our experience, the transition becomes much smoother if, from the very beginning, we explain to the students that a Turing machine is nothing else but a finite automaton equipped with *two* stacks.

This way, the transition from pushdown automata to Turing machines becomes as natural as the transition from finite automata to pushdown automata:

- Since finite automata are not sufficient, we equip them with a stack.
- Similarly, when a finite automaton with a single stack is not sufficient, we equip it with two stacks.

The fact that Turing machines are universal computers – i.e., that they can compute any computable function – indicates that two stacks are enough: if we add three or more stacks, we do not further increase the class of tasks that can be computed by the resulting computational device.

In what sense is a Turing machine a finite automaton with two stacks: details. The interpretation of a Turing machine as a finite automaton with two stacks is easy to explain:

- Everything on the tape before the head is the first stack, with the symbol immediately preceding the head on top.
- The symbol in the cell to which the head points and the symbols in all the following cells form the second stack, with the symbol-to-which-the-head-points on top.

Example: idea. Let us illustrate this idea on the example of a simple Turing machine that adds 1 to a binary number.

In a computer, numbers are usually represented least digit first – since all arithmetic operations start with these digits, and we want to start performing an operation as soon we start accessing the number. From this viewpoint, e.g., the decimal number 6, which is 110 in binary code, is represented as 011. It is reasonable to follow this idea when representing binary numbers on a Turing machine. For example, 6 will be represented as

| | 0 | 1 | 1 | | …

The very first cell is usually left empty, to make sure that the machine knows where the tape starts and does not "fall off the cliff" by trying to get too far to the left. Computations usually start with the head pointing to the very first cell. At the end of the computations, we expect the head to be back in the starting position.

In this representation, adding 1 is easy:

- we move right;
- if we see 1, we replace it with 0 and continue moving right.
- If we see 0 or blank, we replace it with 1, and then rewind, i.e., go back until the head points to the very first cell.

One can easily check this on the example of adding 3 and 1:

```
 - - -
   11
  + 1
-------
  100
```

First, we add the last 1 of the given number and the 1 that we are adding. As a result, we get 2, i.e., 10 in binary code. This means that we get 0 and get 1 carry. Then, we add the carry 1 to the last-but-one digit and again get 10, i.e., 0 and 1 carry. Finally, we get the carry only, so we place 1.

Example: a formal description in the Turing machine terms. In the Turing machine terms, we can consider four states:

- The starting state "start",
- The state "move", when we move to the right.
- The state "back", when we move back.
- The state "halt", when we stop.

We have the following rules in which:

- _ means empty cell,
- R means going right, and
- L means going left:

start, _ → move, R
move, 1 → 0, R (meaning that we replace 1 with 0 and go right, without changing the state)
move, 0 → back, 1, L
move, _ → back, 1, L
back, 0 → L
back, _ → halt

Tracing the Turing machine. Let us illustrate these rules on the example of adding 1 to 3 (i.e., to 11 in binary code). In this sequence of operations, the state of the head is illustrated by the first letter of the state's name: s for start, m for move, etc. We start with the state in which 11 is written on the tape, and the head point to the first (empty) cell:

```
|  | 1 | 1 |  | …
 s
```

The Turing machine is in the state "start" and it sees an empty cell. So, according to the first rule, it changes its state to "move" and the head moves one step to the right:

```
|  | 1 | 1 |  | …
    m
```

Now, the head is in the state "move" and it sees 1. So, according to the second rule, it should replace 1 with 0 and move to the right:

```
|  | 0 | 1 |  | …
      m
```

The same rule is applied again, so we get

```
|  | 0 | 0 |  | …
        m
```

Now, we use the fourth rule: replace blank with 1, change to the state "back" and go one step to the left:

```
|  | 0 | 0 | 1 | …
      b
```

We see 0, so we continue moving to the left until we see blank, at which point we halt:

```
|  | 0 | 0 | 1 | …
    b

|  | 0 | 0 | 1 | …
 b

|  | 0 | 0 | 1 | …
 h
```

Example reformulated in two-stacks terms. Let us show how all these states look like in terms of the two stacks – and how each transition can be represented by pushing and popping of these stacks. We will represent each stack as a—b—c-- …, where a is the element at the top of the stack, b is next to it, c is next, etc.

In the starting state, there is nothing before the head, so the first stack is empty, and the second has blank followed by two 1s:

```
|  | 1 | 1 |  | …     Stack 1: empty,     Stack 2: --1—1--
 s
```

In the next state, the first stack contains the blank symbol, and the second stack contains only the two 1s:

```
|  | 1 | 1 |  | …    Stack 1: --,    Stack 2: 1—1--
    m
```

To get to this state, we pop the top symbol from the second stack (in this case, the blank) and push it into the first stack. This is a typical description of what happens when the head moves one step to the right. We continue:

```
|  | 0 | 1 |  | …    Stack 1: 0--  --,    Stack 2: 1—
       m
```

To get to this state from the previous one, we:

- first replace 1 with 0, i.e., pop 1 and push 0 instead, and
- then again emulate moving right, i.e., pop the top symbol (0) from the second stack and push it into the first stack.

After that, similarly, we get

```
|  | 0 | 0 |  | …    Stack 1: 0--0--  --,    Stack 2:  —
          m
```

In addition to what we have done earlier, we also need to push the blank symbol into the second stack, to indicate that this blank symbol is what the head sees. Such addition is needed because the tape is infinite, we can have as many cells as we want, but the stacks are finite, so when we get to the place on the tape that was never used before, we need to add the corresponding cell to the tape.

Now, according to our rules, we replace blank with 1, after which the head will start moving left. We already know how to replace. Moving left means that we pop the top symbol from the first stack and push it into the second stack. As a result, we get the following:

```
|  | 0 | 0 | 1 | …   Stack 1: 0-- --,  Stack 2: 0—1—
```

b

After this, we continue moving left:

| | 0 | 0 | 1 | … Stack 1: --, Stack 2: 0—0—1--
b

| | 0 | 0 | 1 | … Stack 1: empty, Stack 2: --0—0—1--
b

| | 0 | 0 | 1 | … Stack 1: empty, Stack 2: --0—0—1--
h

Acknowledgments. This work was supported in part by the US National Science Foundation grant HRD-1242122 (Cyber-ShARE Center).

The authors are thankful to Dr. Mourat Tchoshanov for his encouragement.

Reference

1. Sipser, M. Introduction to the Theory of Computation, PWS Publishing Co., Boston, Massachusetts, 2006.